Java集合：Collection+Map(都在java.util包下)

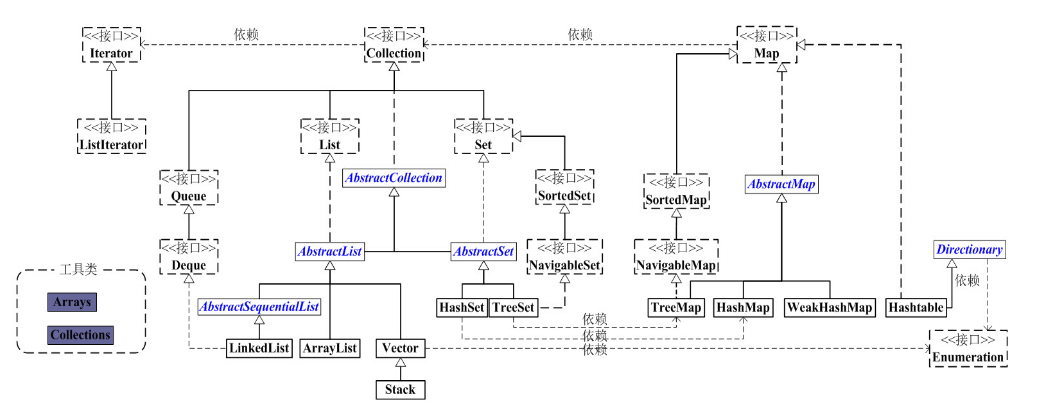
Collection是一个高度抽象的集合接口其子接口有：

List：可重复的有序队列，常见实现类ArrayList，LinkedList，Vector，Stack(数据结构中的链式存储)

Set：不可重复的无序集合，常见实现类HashSet，TreeSet

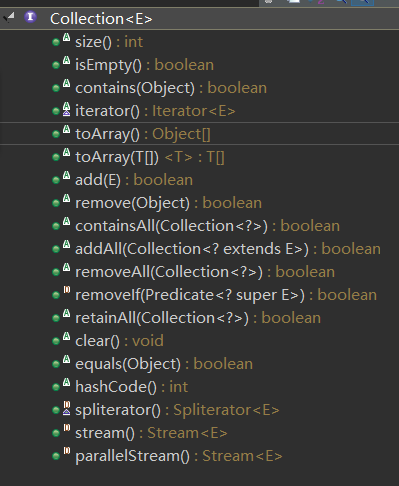
Queue：队列被继承(数据结构中的队列)

Map是K-V的集合，常见实现类HashMap，Hashtable，LinkedHashMap



Collection：

Collection是一个高度抽象的集合接口，包含了集合的基本操作(CRUD，遍历等)

Collection还继承了Iterable接口用于Iterator的遍历

Collection只是接口，具体的方法实现还是需要类的，AbstractCollection抽象类对Collection中的大部分方法进行了实现，除了iterator和size方法，这样方便其他直接继承即可

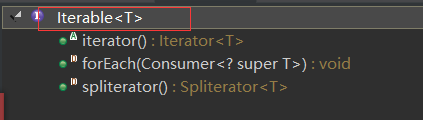
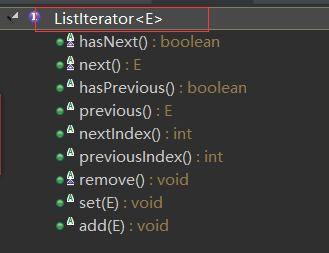
List：List接口继承Collection，并且定义了一些List特有的方法，List代表的是一个链式存储，同样List也有一个AbstractList实现了List相关的方法，但是AbstractList直接继承了AbstractCollection类和实现了List接口，实现AbstractList的最大作用就是实现List相关的方法，其他实现类直接使用即可，实现List接口是为了实现List特有的方法并且声明该类是一个List，在AbstractList声明了两个内部类







ListIterator是List专门遍历的接口，声明了一些List特有遍历的方法(向前/后遍历，是否存在上一个元素等，毕竟List是一个链式存储)，并且在内部类中声明了方法的使用

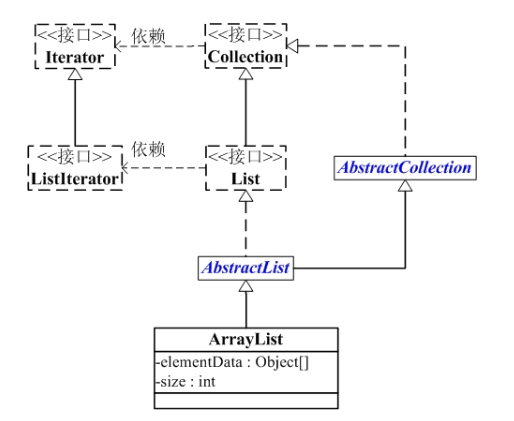
 

ArrayList：

ArrayList是List常用的一个实例类，是一个数组链式存储，通过数组实现的

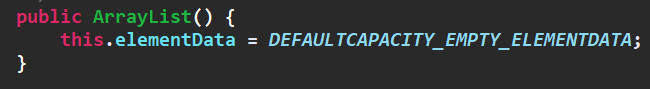


这里有个小问题：AbstractList就是List的实现类，而这里又实现List，多余了，有人认为这当初为了后期的维护所设计的，实不实现List接口都无所谓(这里知道有这个即可)，RandomAccess是用于随机访问(即可以通过元素的序号快速的获取到元素就叫随机访问)，Cloneable表示当前类可以克隆，只有实现了Cloneable接口才能调用Object的clone()方法



ArrayList不是线程安全(非同步)的，不能在多线程中使用

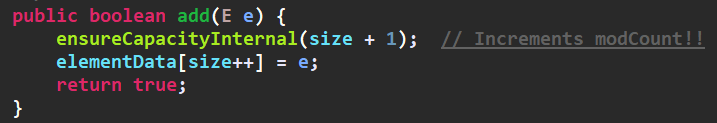
创建ArrayList：Object[] elementData是存储ArrayList的数组结构



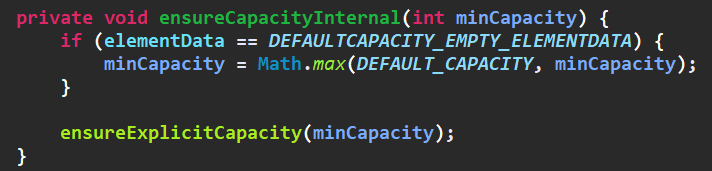


在调用无参方法创建ArrayList时会将一个空的字符串赋值给elementData(基于JDK1.8)，此时elementData为空，大小为0

在第一次调用add方法的时候会初始化一个大小为10的数组

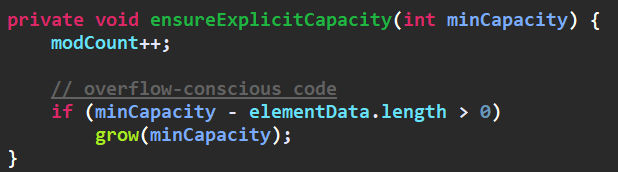


这里的size是elementData在元素的个数(不是数组的大小)，首先要确保elementData中是否还能装元素

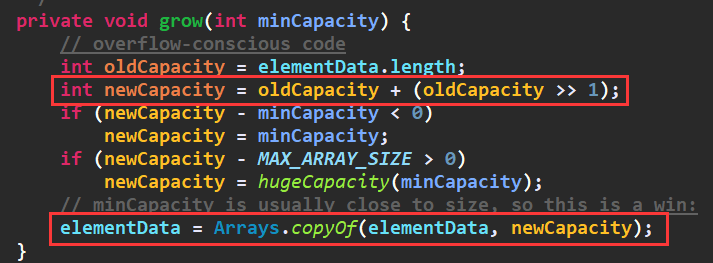




先判断是否是第一次向elementData中添加元素(即elementData == *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*)，初始化数字的大小为10，如果是第二次添加，直接执行ensureExplicitCapacity判断当前elementData还能不能装元素



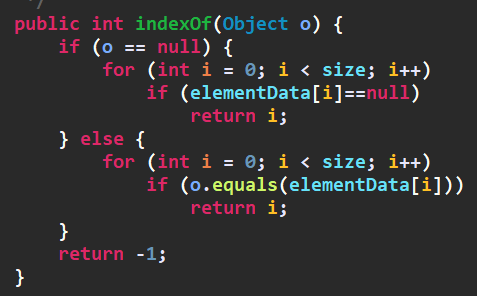
这里的modCount是用于Iterator操作的(后面会讲，modCount：modify Count)，这里判断当前elementData还能不装元素，如果不能，对elementData进行动态添加(grow，此时的minCapacity表示的就是当前elementData中元素的大小)



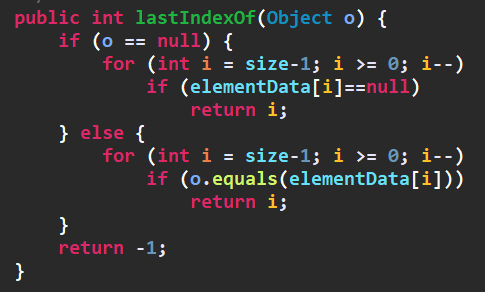
将数组的大小增加到原来的一半，并且将久元素复制到新的elementData中，这样就实现了对ArrayList的动态增长

获取ArrayList元素的方法：

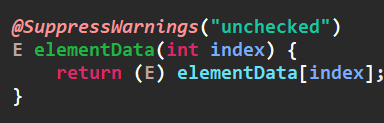
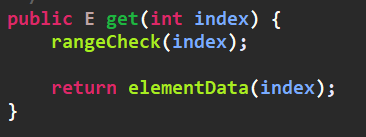
根据Object获取元素 下标，正序获取第一个出现的元素的下标



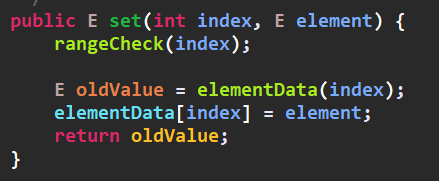
逆序获取指定元素的下标



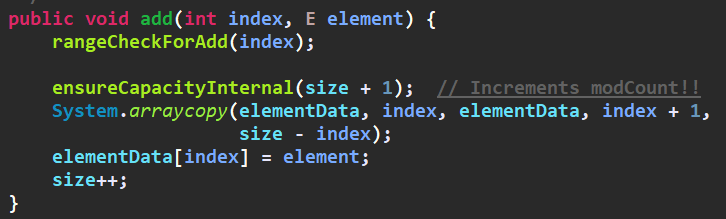
通过下标获取元素



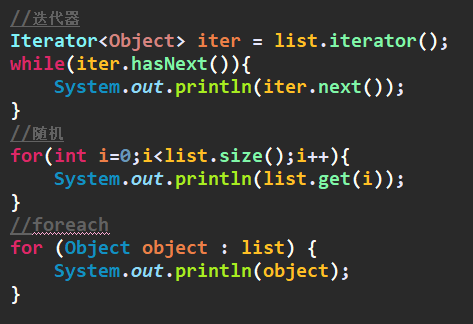
修改指定下标元素，并返回久元素



指定位置添加元素(数组实现)



遍历ArrayList的方法：



从访问效率来讲，随机更快，foreach次之，迭代器最差，其实foreach底层还是通过迭代器实现的

ArrayList中Iterator的实现

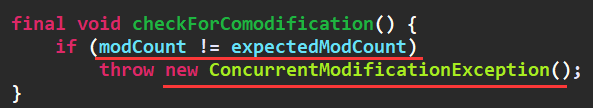
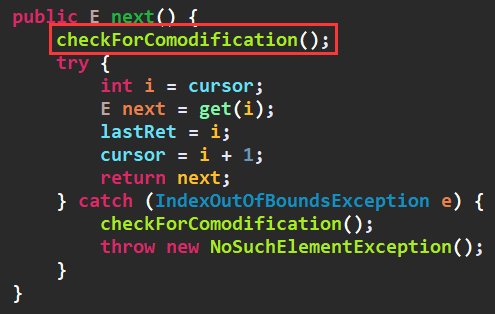
fail-fast机制：

当线程A遍历ArrayList，线程B向ArrayList中添加元素，因为ArrayList是非同步的，此时就会抛出ConcurrentModificationException的异常，fail-fast机制只会出现在通过Iterator遍历集合时，又添加元素的时候

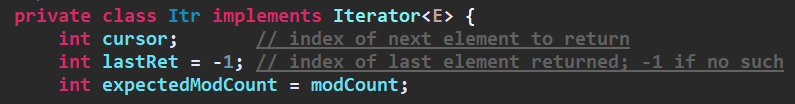
fail-fast的实现机制：在前面我们讲到了在对ArrayList的方法进行操作时，会对modCount进行自增，在ArrayList内部声明了一个内部类，实现了自己的Iterator，并且将modCount赋值给expectedModCount，每次进行Iterator的相关方法时，都会检验expectedModCount和modCount的值是否相等，如果只是对ArrayList进行简单的遍历是不会对modCount进行修改的，只有当前其他线程对ArrayList进行添加或删除才会对modCount进行修改，修改后就会导致expectedModCount和modCount不相等，导致抛出异常





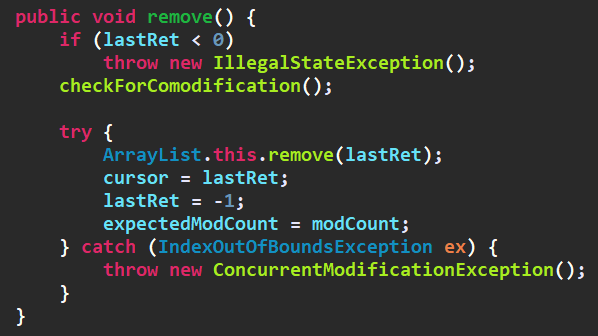
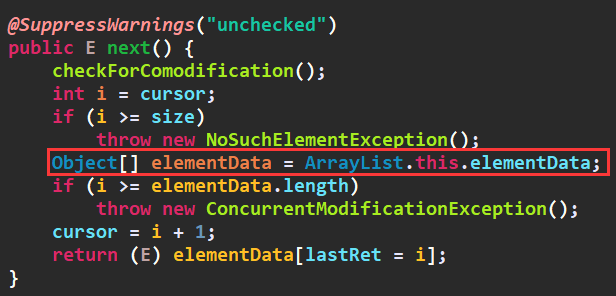


Iterator的方法



cursor：当前返回元素的下标

lastRet：最后一个元素的下标



next

首先检测expectedModCount和modCount是否相等

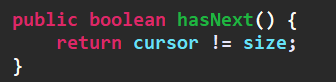
这里每次获取ArrayList的原因：当线程A访问到倒数第二个元素，线程B删除了最后一个元素，这样就会抛出异常，而此异常属于ConcurrentModificationException异常，为了捕获此异常，才为了每次都去获取

Remove

在使用迭代器遍历元素，然后对指定元素删除时，不能直接调用List的remove方法，当在调用迭代器的next等方法时，就会抛出ConcurrentModificationException异常，这里可以采用荣迭代器的remove方法(fail-fast机制只会在迭代器中出现)

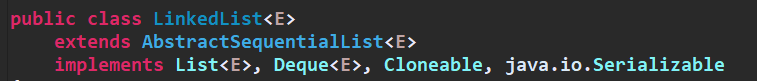
Remove和next是相互依存的，必须先执行了next才能调用remove，而且这里的remove实际上还是调用的是ArrayList的remove，然后再执行expectedModCount = modCount

这是下标问题，这里删除的是迭代器刚越过的元素

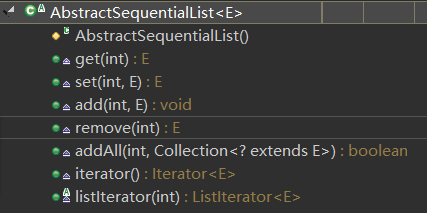


当前下标和elementData的size是否相等

LinkedList：ArrayList是通过数组实现的链式存储，Linked是通过双向链表实现



LinkedList继承AbstractSequentialList，实现Deque接口所以也可以将LinkedList作为队列来使用



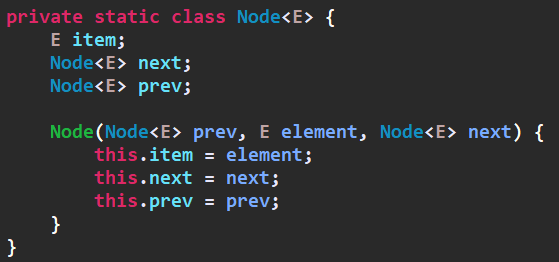
LinkedList是一个双向链表，在AbstractSequentialList中实现了一些随机访问的方法(随机访问针对的是数组，可以通过下标直接访问，而顺序访问针对的是链式存储，顺序结构)，AbstractSequentialList继承于AbstractList，所以LinkedList直接继承AbstractSequentialList即可

LinkedList还实现了Deque接口，所以可以将LinkedList视为双端队列(Deque继承于Queue)



Queue只是一个队列，先进先出

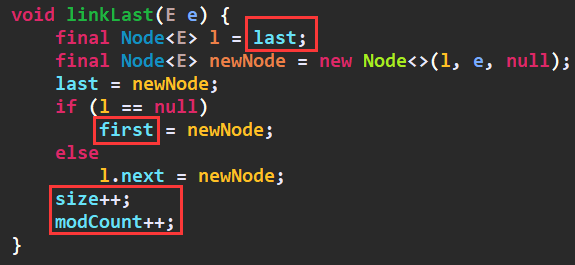
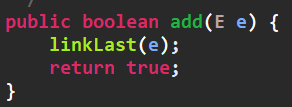
Deque是一个双端队列，实现的方法更多



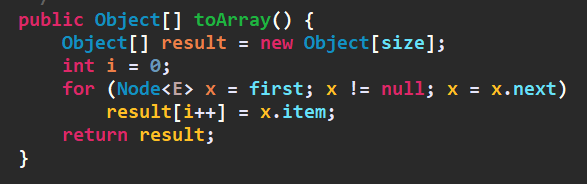
定义节点

new LinkedList()时，只会创建一个ArrayList，不会创建头尾节点

只有在添加的时候才会创建头尾节点



size表示当前元素个数

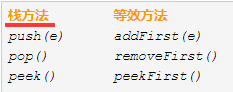


双向链表能操作的在LinkedList中都能找到对应方法

LinkedList是不会存在容量不足的问题

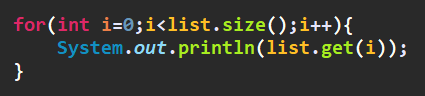
在LinkedList中还实现了队列(Queue)的相关操作(pop,peek,push等)

LinkedList不仅可以实现先进先出的队列，还能实现先进后出的栈

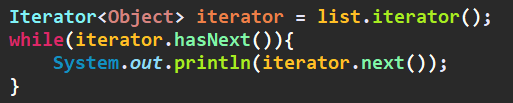


LinkedList的遍历

LinkedList顺序访问的速度比随机访问的速度快



调用get还是去遍历查询

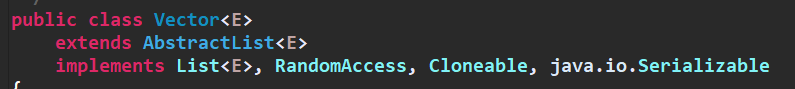


这里的Iterator实际上是LinkedList内部类中的ListIterator

还可以通过removeFirst/removeLast/pollFirst/pollLast依次从链表中取出对象(这里的取出就移出)

由于ArrayList和LinkedList都是非同步(非线程安全)，所以不适用于多线程环境下，而List下的Vector和Stack是线程安全的

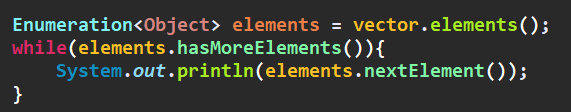
Vector：



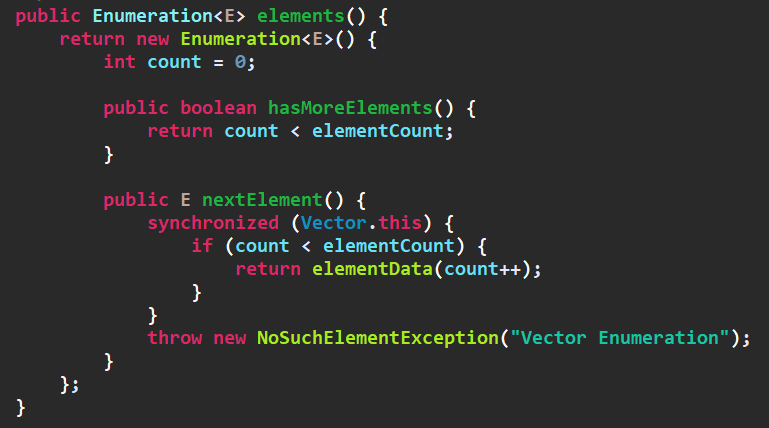
通过Vector继承的类和实现的接口，可以看出Vector和ArrayList差不多，都是通过数组实现的，Vector是一个**队列**，先进先出并且是同步，每个方法都被synchronized修饰

Vector的是用方法基本和ArrayList差不多

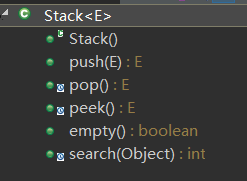
遍历：迭代器(底层还是通过数组下标获取元素的)，随机(get)，foreach，Enumeration



用法和Iterator类似

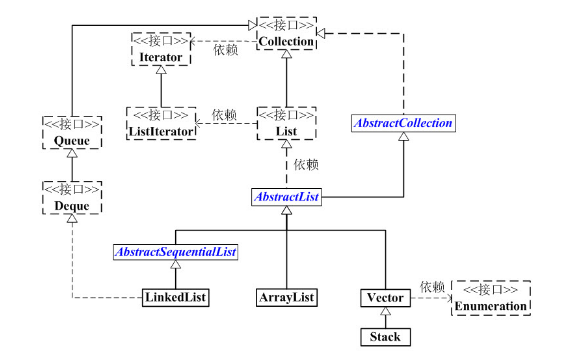


Stack：



Stack是栈，先进后出，通过Vector的数组结构实现，Stack也是线程安全的

List总结



AbstractSequentialList 实现了“链表中，根据index索引值操作链表的全部函数”(实现了链表操作的相关函数)

ArrayList 是一个数组队列，相当于动态数组。它由数组实现，随机访问效率高，随机插入、随机删除效率低。

LinkedList 是一个双向链表。它也可以被当作堆栈、队列或双端队列进行操作。LinkedList随机访问效率低，但随机插入、随机删除效率低。

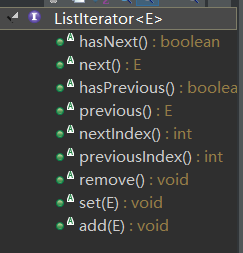
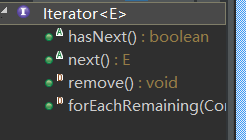
Vector 是矢量队列，和ArrayList一样，它也是一个动态数组，由数组实现。但是ArrayList是非线程安全的，而Vector是线程安全的。

Stack 是栈，它继承于Vector。它的特性是：先进后出。

涉及栈，队列，链表的操作都采用list，具体使用哪个List根据查询多还是修改多来判断

Iterator和ListIterator的区别：

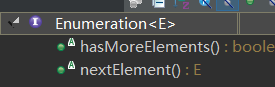
Iterator是通用的遍历集合的接口，而ListIterator根据List的特征声明了一些适合List的方法



注意：这里的remove是移出的是当前元素的下一个元素，实际上还是调用具体集合类的remove方法

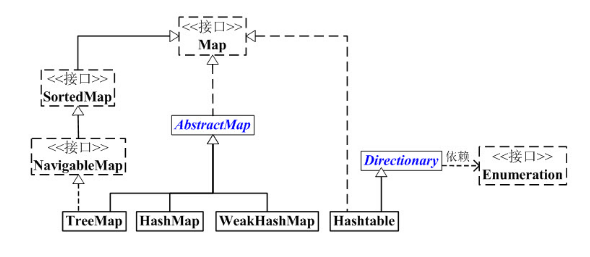
Iterator和Enumeration的区别：

Iterator和Enumeration都是用于遍历集合



Enumeration只声明了遍历的方法，没有声明移出的方法，而且Enumeration只能在HashTable，Vector，Stack中使用，Enumeration使用得很少了，Iterator就是代替Enumeration

Map



和Collection一样，Map接口声明Map通用的方法，AbstractMap对这些方法进行实现，被其他类继承

SortedMap中的内容是排序的键值对，是有序的

NavigableMap对SortedMap进行实现，并且实现一些其他方法

TreeMap中的内容是有序的键值对，而HashMap中的内容是无序的键值对

HashTable中的内容是无序的键值对，且和HashMap相比，Hashtable是线程安全的

WeakHashMap的键是弱键

Map映射中不能包含重复的键，每个键只能对应一个值

Map提供三种视图获取映射关系

entrySet()：获取K-V的Set集合



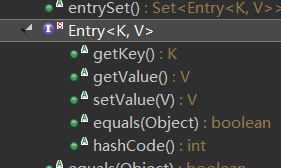
keyset()：获取K的set集合



values()：获取V的Collection集合



在Map接口内部定义了一个操作K-V的接口Entry



entrySet获取的Set集合装的就是Entry

先看看Hash表(散列表)

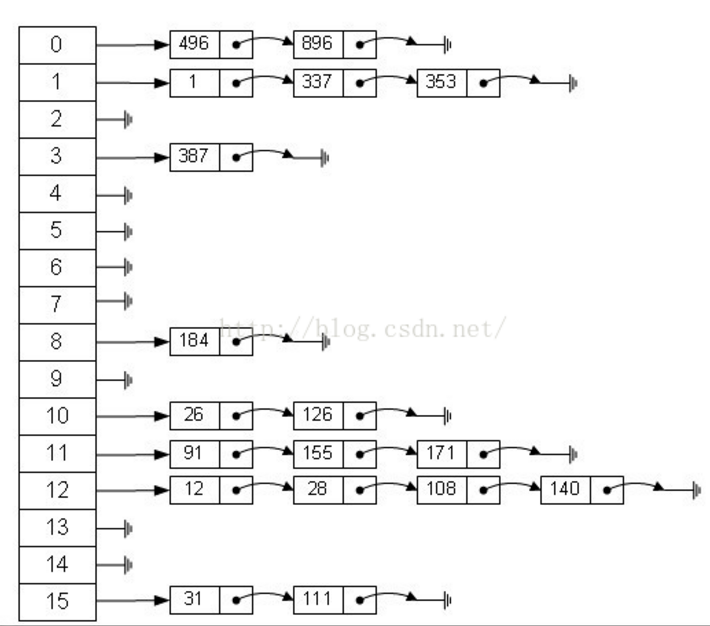
数组的特点是：寻址容易，插入和删除困难

链表的特点是：寻址困难，插入和删除容易

而Hash表兼容了两者的特点，Hash的数据结构是，先声明一个数组保存数据，K-V结构的数据，当输入K时，通过Hash函数对K进行编码，然后做相关的运算，计算出一个数组的下标，将当前V保存在此数组，这样就将K-V的数据结构的V保存在数组中了，这种保存不是顺序存储，而是散列在数组的不同位置，但是会存在空间浪费的情况

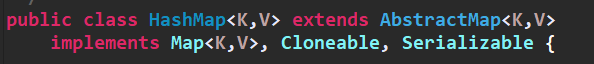
但是当K1！=K2，通过hash编码后Hash(K1)=Hash(K2)，这叫Hash冲突，解决这种冲突常见的方法就是拉链法，即在每个数组后定义一个单向的链表，当Hash冲突时，将相同Hash保存在数组对应下的链表中

在查询数据时候输入K，同Hash将K进行换行找到对应数组的下标，然后对数组对应的链表进行遍历查询

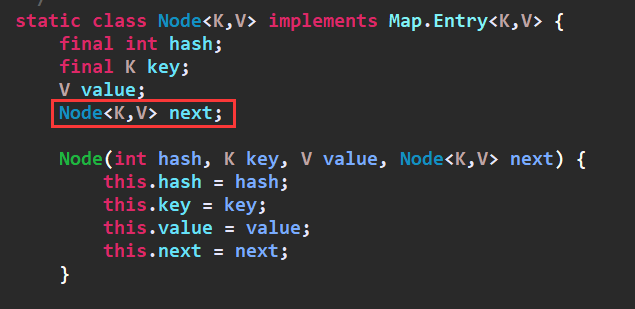


负载因子：在创建Hash表的时候会声明一个负载因子，理论上来说，声明一个大小为10的数组即可不用对Hash表的容量进行扩展，因为链表会自动扩展，但是这样又和定义一个链表有什么区别，所以当Hash表中元素达到一定程度的时候就要对其扩充，负载因子就是什么时候进行扩展的指标，当元素超过(数组容量\*负载因子)时，就对其数组大小进行扩充，一般默认的负载因子为0.75，这里小于1的原因是，因为通过Hash计算后，数组被全部填满的概率很小，更多情况下时重复。0.75是经过大量数据计算出来的，如果过大，那么元素大多以链表的形式存在，如果过小，那么元素大多以数组形式存在。同样可以根据自己的需求(查询多/修改多)来设置赋值因子的大小

HashMap：非线程安全，无序K-V







在Hash内部声明了一个Node数组，Node是HashMap的一个内部类，Node<K,V> next表明HashMap采用的拉链法解决Hash冲突

HashMap中几个重要的成员变量

transient Node<K,V>[] table：Hash表的数组

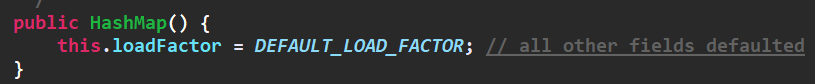
transient int size：Hash表中元素的个数

int threshold：HashMap的阀值，就是负载因子\*数组容量

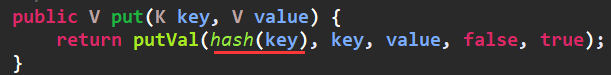
final float loadFactor：负载因子

transient int modCount：fail-fast机制

new HashMap()：创建HashMap的时候可以通过无参构造器，也可以指定capacity，负载因子的大小，一共4个构造函数



这里只是声明了负载因子为默认大小，而并未创建数组(jdk1.8创建集合的操作都是先声明然后在调用put/add方法的时候再创建集合对应的数据结构)

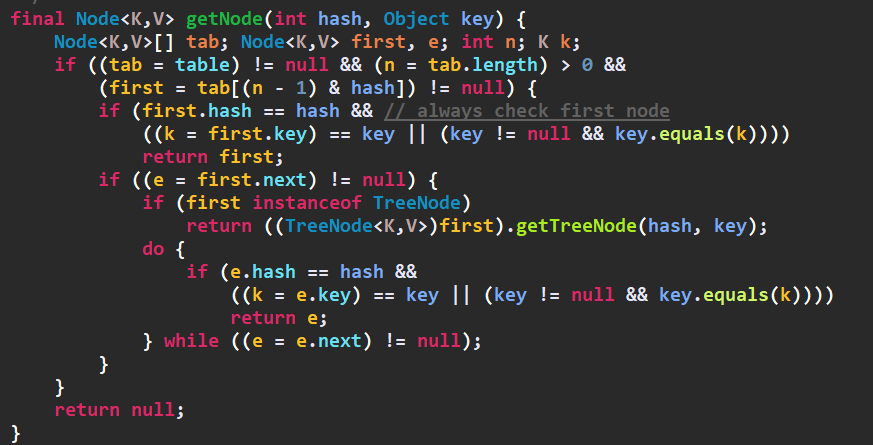


会去判断table(即Node的数组结构)是否为空，如果为空，调用resize函数创建一个初始化默认大小为16的数组(在创建HashMap的时候，数组的大小必须为2的幂次方，在向Hash表中插入键值对时，需要找到数组的下标，此时就需要进行运算

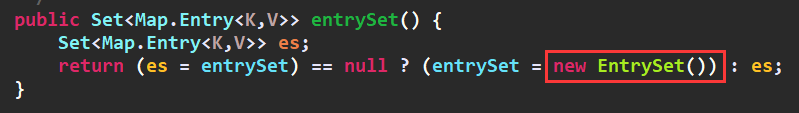
table[(size-1)&hash]==null时，&运算是二进制运行，size是2的幂次，size-1的二进制就是1111…111，在于hash进行&运算时，速度非常快(0&0=0，0&1=0,1&0=0，1&1=1))，然后将创建好的table返回，将K-V插入到数组中

通过查看源码，可以看到一个TreeNode，JDK1.8的HashMap实际上采用的数组+链表+红黑树的数据结构，当链表的长度超过8，HashMap会自动将Node插入到红黑树中

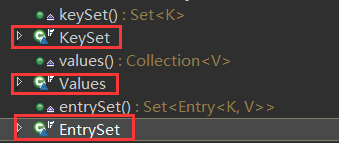
当插入完一个Node后，HashMap才会去检测当前HashMap的size超过threshold，是否对HashMap进行扩容(注意是添加之后)，扩容是扩大到原来的2倍



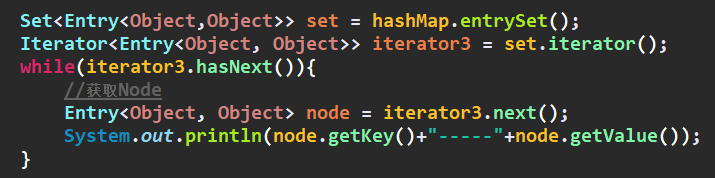
entrySet：获取K-V



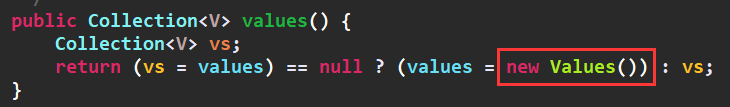


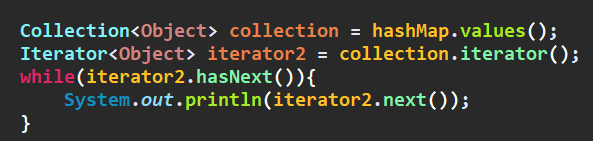


在Map内部为每个获取K,V提供了一个内部类，用户封装获取的结果，并提供了一些方法

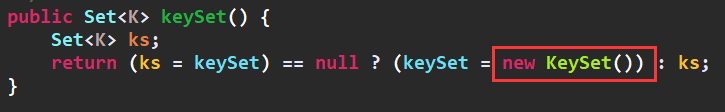


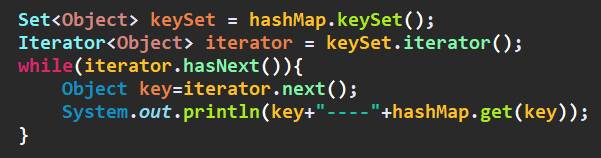
values：获取V



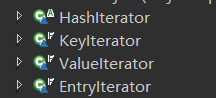


keyset：获取K



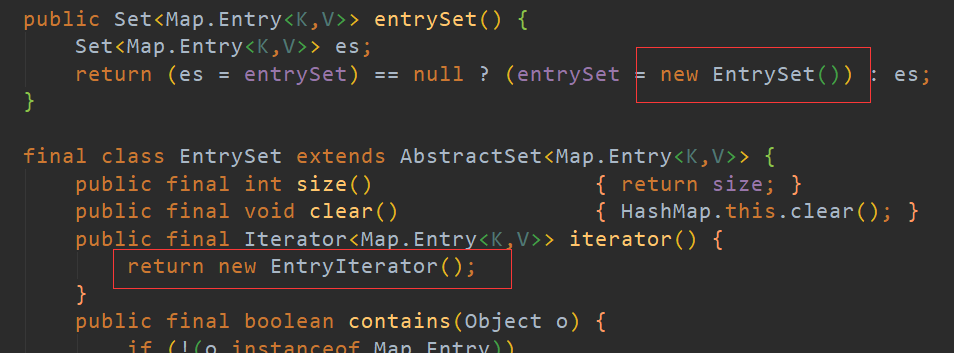


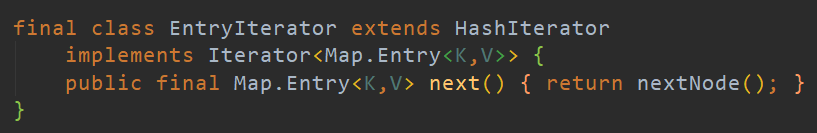
遍历的方式：K,V,K-V

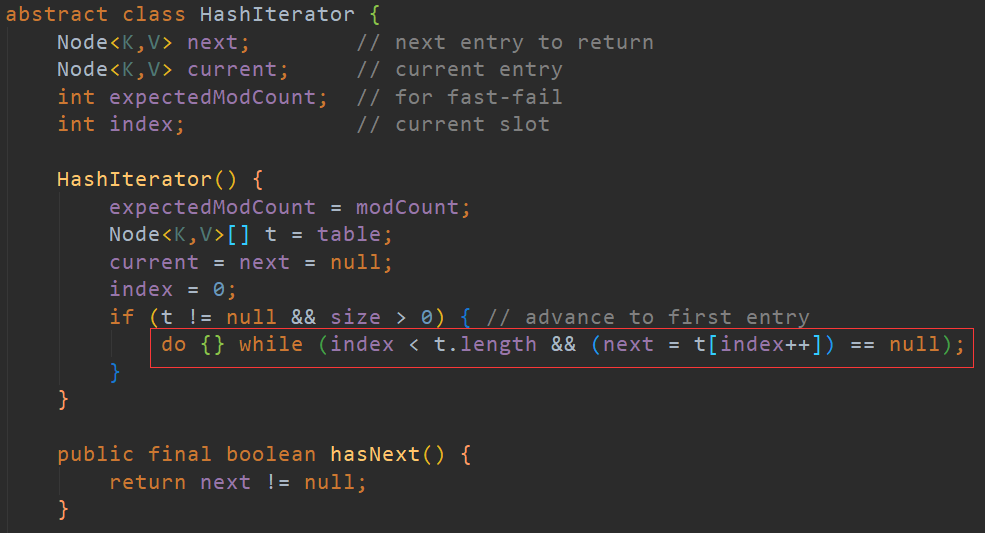


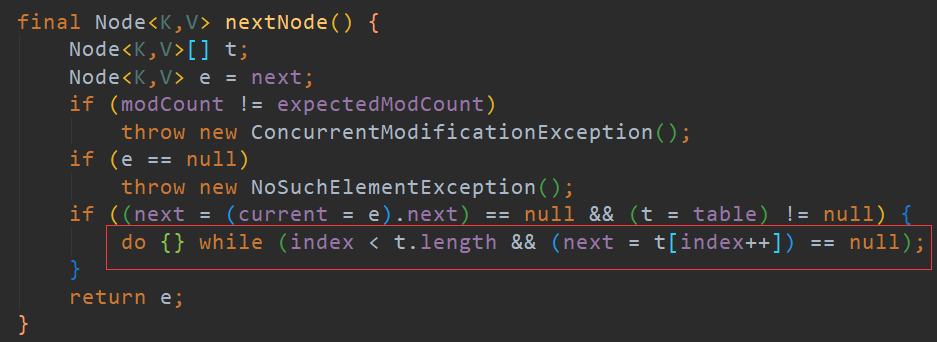
containsKey/containsValue：判断是否存在指定键或值，返回值是Boolean值

Map虽然未实现Iterator接口，但是却声明了用于fail-fast的ModCount，主要是在对K，V，K-V进行迭代的时候使用，实现方式：在调用entrySet时，会创建实例化一个EntrySet类(HashMap内部的)，然后在EntrySet构造器中创建了一个**EntryIterator**，而EntryIterator又继承**HashIterator**，在HashIterator对HashMap的K-V迭代实现







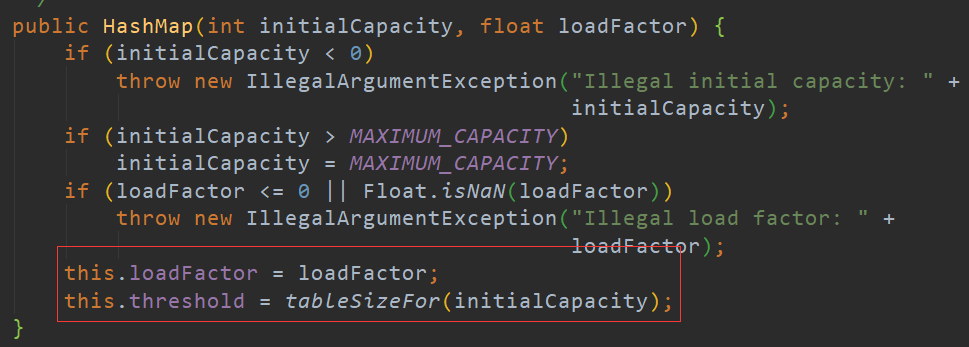


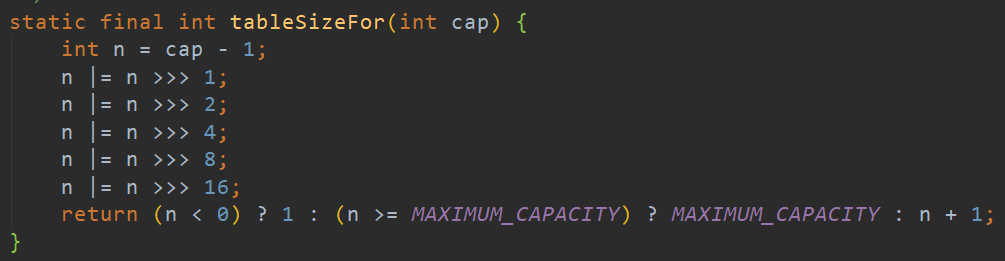
在HashIterator的构造器中先找到第一个元素(next)，在执行HashIterator的hasNext方法，然后在调用EntryIterator的next方法，也就是HashIterator的nextNode方法，找出下一个元素(注意do-while循环)，对K和V的迭代实际上都是基于K-V的

注意：迭代的hashNext和next方法的判读都是在原来集合上进行判断的，而不是将原来集合复制到Iterator中再进行判断，这样也不符合实际

HashMap链表长度超过8时，就会将链表中的节点转换为红黑树

HashMap初始化大小的确定：(存储元素个数/负载因子) + 1

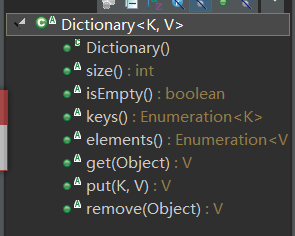
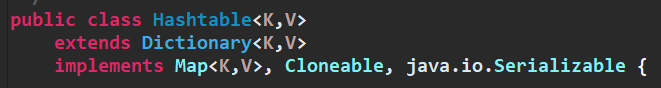




初始化时自定义HashMap的大小必须是2的幂次，而tableSizeFor就是对输入的值进行转换，如果输入值是2的幂次，返回值为输入值，如果输入值不是2的幂次，返回值为大于输入值的2的幂次，输入值为9，返回值为16；如果没有cap -1 ，输入2的幂次时，返回值将是输出值的2倍，所以要有个cap – 1

Threshold为HashMap的阀值，在执行完tableSizeFor，threshold的值任然为输入的初始化值，而阀值的计算再resize()扩容函数中，第一次put的时候会扩容(初始化)

Hashtable：

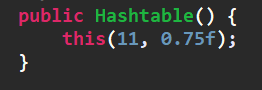


Hashtable并未继承AbstractMap，只实现了Map接口，Dictionary抽象类中声明了一些抽象方法并未实现

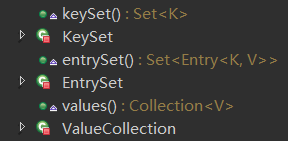
Hashtable的实现方式和HashMap的方式一样，都采用的hash表，但是是采用数组+链表(Hashtable和以前的版本没有多大区别)

Hashtable是线程同步的(HashMap非同步)，并且Hashtable和HashMap一样，元素都是无序的，而且Hashtable中的元素不能为null(HashMap可以)，因为HashTable是线程同步，如果get到一个null值，不知道是这个key不存在，还是其他线程将这个key的value设置为null，所以HashTable直接不允许为null

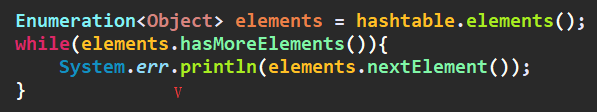
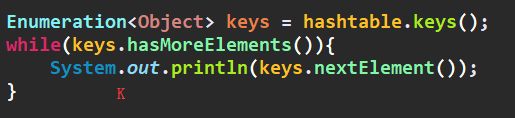
Jdk1.8的Hashtable初始化容量是11，负载因子为0.75，扩容为old\*2+1



遍历方式



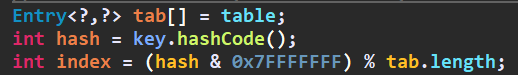
和HashMap一样，在通过Iterator遍历K,V,K-V，Hashtable还实现了Enumeration接口，还可以通过Enumeration遍历



进行扩容时，都是创建一个新的，然后将原来的赋值给新的

Hashtable和HashMap通过Hash计算索引的方式不同(这就是为什么初始化长度不一样的原因)

Hashtable



HashMap：h为hash值，n为数组的长度





HashMap的默认大小为16，而且在增加是原容量\*2

Hashtable默认大小是11，增加是原始容量\*2+1

Hashtable主要是线程安全吗，而HashMap是非同步的

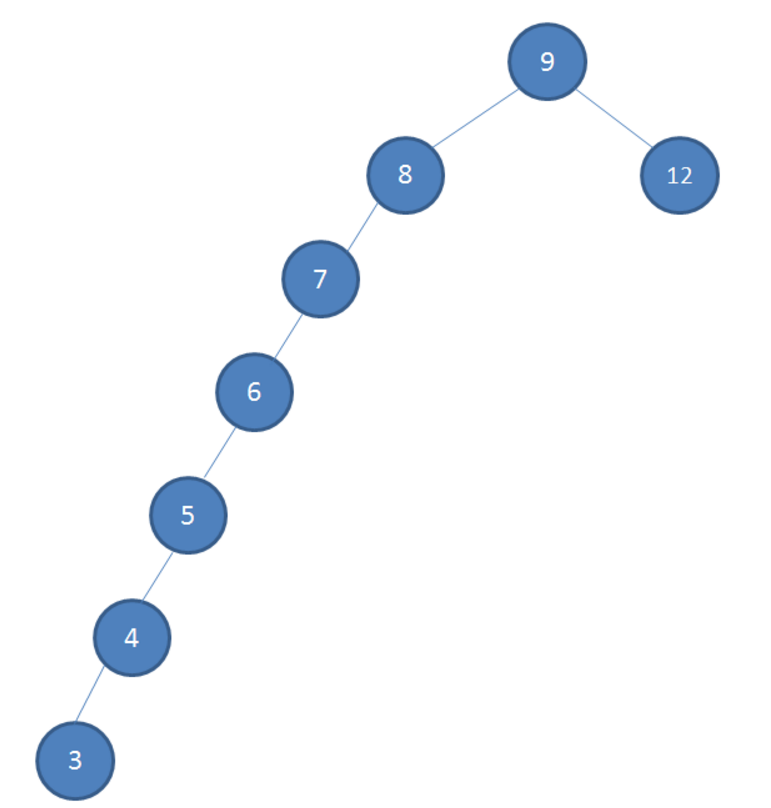
HashTable继承Dictionary，HashMap继承abstractMap

HashMap在遍历的时候是从前向后遍历数组，Hashtable是从后向前遍历数组

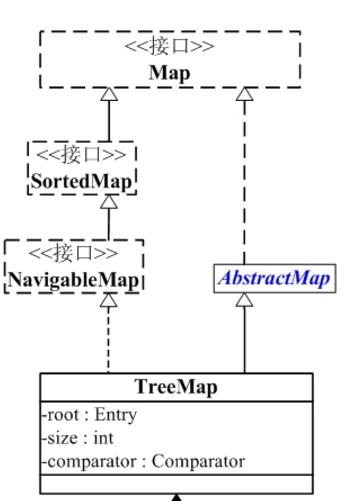
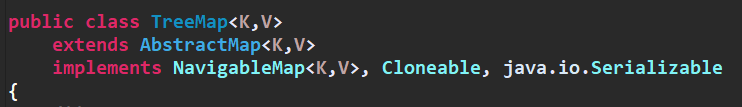
TreeMap：

在介绍TreeMap之前，先介绍一下红黑树，在介绍红黑树之前在来说说平衡二叉树，平衡二叉树有很多优点，但是当我们插入一段连续的数字时，就会导致树不平衡，和链表差不多，所以引入红黑树来保持树的平衡(保持平衡的方法：变色，左旋/右旋)

<http://www.sohu.com/a/201923614_466939>



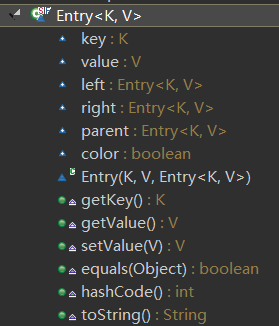
TreeMap的底层就是采用红黑树实现的



SortedMap接口中的内容是排序的键值对，排序方法是通过比较器(comparator)

NavigableMap提供了导航的方法，如获取大于/等于某对象的键值对，小于等于某对象的键值对(树)

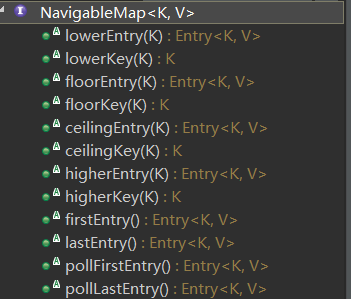
TreeMap是有序的集合，因为是通过树实现的，TreeMap是非同步的



定义红黑树的节点

TreeMap的操作和HashMap类似，获取K,V,K-V操作

NavigableMap提供导航方法获取K,K-V



K：

firstKey()/lastKey()获取树的最左端/最右端

lowerKey()/higherKey()获取当前key的上/下一个

floorKey()/ceilingKey()获取大于等于/小于等于当前Key

K-V：

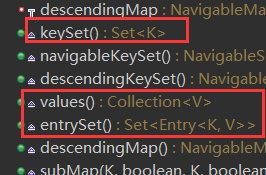
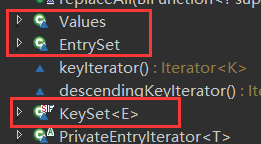
firstEntry()/lastEntry()/lowerEntry()/higherEntry()/floorEntry()/ceilingEntry()

pollFirstEntry()/pollLastEntry()：获取第一个/最后一个entry，并删除

在TreeMap中对这些方法进行了实现

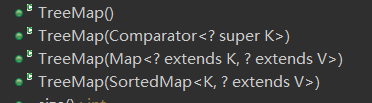
这些只是获取指定K，K-v

在TreeMap中也提供了通过迭代器获取K,V,K-V集合的方式，声明内部类，实现方法



遍历方式也是通过Iterator对K,V,K-V进行遍历

descendingMap/descendingKeySet：方向获取Map和key



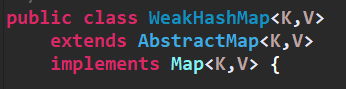
TreeMap提供了4个构造函数，Comparator传入自定义的比较器，当执行stored时，会调用Comparator的compareTo方法，传入自定义的Comparator可以根据自定义的规则进行排序

TreeMap无参构造器使用默认的比较器，传入的Map的因为是无序的，所以要依次输入到红黑树中，SortedMap是有序的，但是在转化为红黑树时还是要做相关的变化

TreeMap.headMap(K)：获取大于K的集合，默认不包含K，可以添加参数是否包含

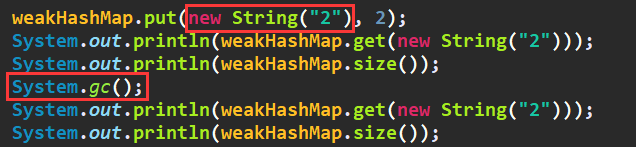
TreeMap.tailMap(K)：获取小于K的集合，默认包含K，可以添加参数是否包含

WeakHashMap：



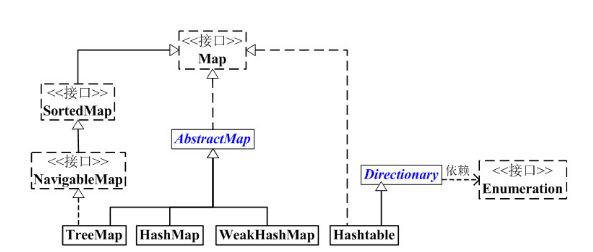
WeakHashMap为实现Serializable接口和Cloneable

WeakHashMap和HashMap一样也是散列表，WeakHashMap和HashMap差不多唯一不同之处在于WeakHashMap的键是弱键(即如果WeakHashMap的键是一个对象，那么当JVM调用GC时，弱键和对应的值将会清除)



这里手动调用gc，调用gc后输出的结果为null 0(Entry为弱键)

在Java中，WeakReference和ReferenceQueue 是联合使用的，弱键的类型是WeakReference，RefererQueue保存被gc回收的弱键，当调用gc时，会将弱键放入到RefererQueue中(弱键必须是对象)。



HashMap是基于拉链法的hash表，用于单线程中，红黑树，元素可以为null

Hashtable是基于拉链法的hash表，线程安全，用于多线程中，元素不能为null

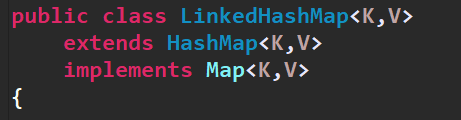
WeakHashMap是基于拉链法的hash表，非同步，弱键引用

TreeMap是有序的，基于红黑树实现，非线程同步

Properties也是一个键值对结构(K和V都是String)，properties实际上是Hashtable的子类

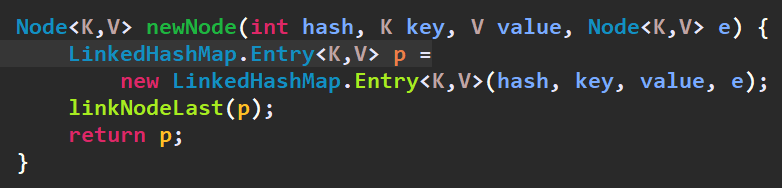


LinkedHashMap是基于HashMap的，是有序的，通过双向链表实现有序

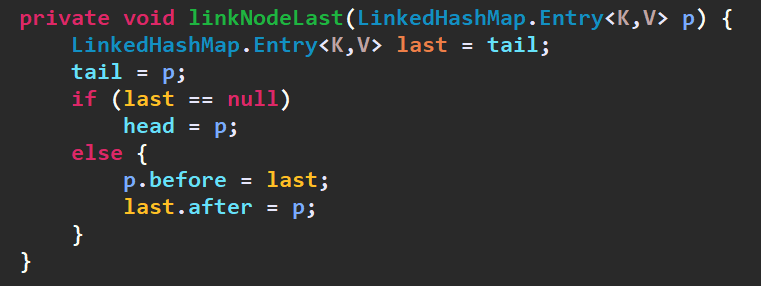


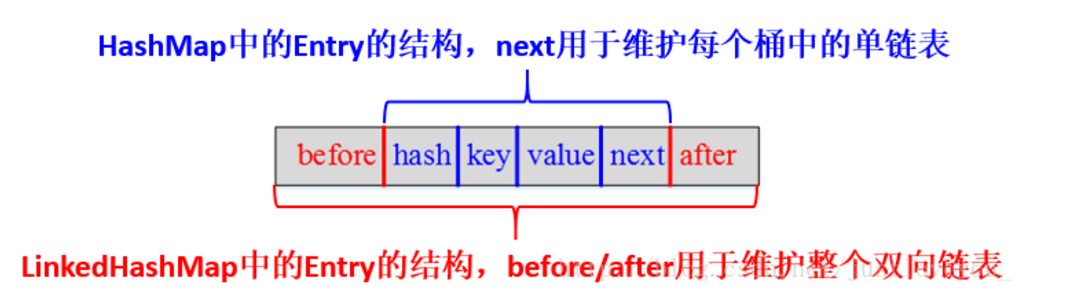
LinkedHashMap通过双向链表保存了出入的顺序，LinkedHashMap和HashMap基本上没多大区别，只是在底层的实现方式上不同

LinkedHashMap并没有put方法，而是使用父类的HashMap的put方法，但是在put方法中调用了LinkedHashMap的newNode创建节点

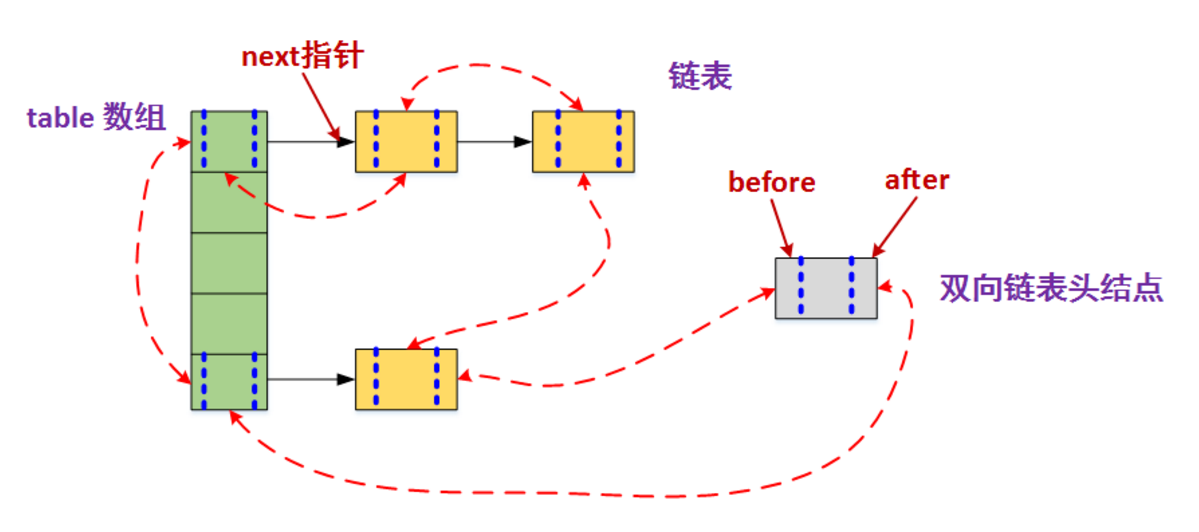


这里的p就是保存到HashMap中的节点值，而linkNodeLast方法是将节点插入到双向链表中





最终LinkHashMap中的节点是这样的

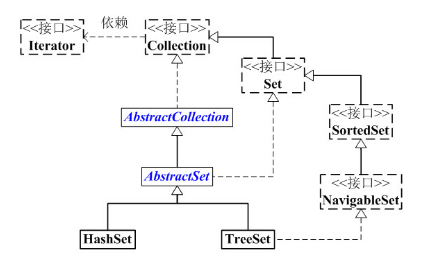


HashMap+双向链表

在读取数据的时候还是通过HashMap的get获取，只是在通过迭代获取的时候，才会去读链表

LinkedHashMap是保存了插入的顺序，TreeMap对插入的值进行了排序操作

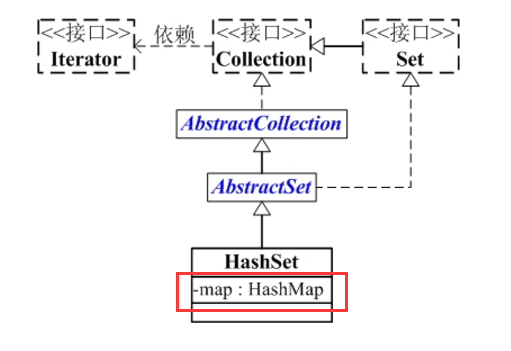
Set：是不允许有重复元素的集合



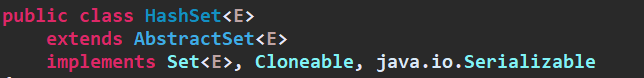
 HashSet依赖于HashMap，它实际上是通过HashMap实现的。HashSet中的元素是无序的。

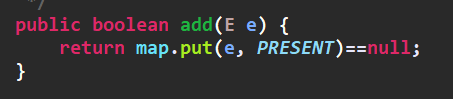
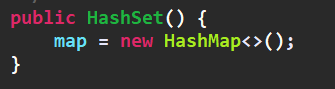
TreeSet依赖于TreeMap，它实际上是通过TreeMap实现的。TreeSet中的元素是有序的。

HashSet：



HashSet是线程不安全的，而且HashSet允许null元素







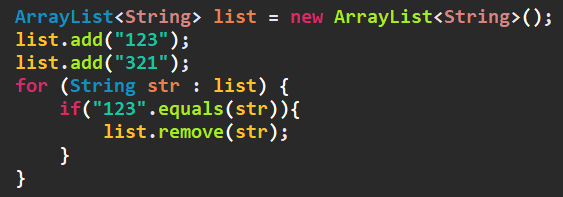
HashSet实际上还是操作的是HashMap，HashSet的值就是HashMap的键(所以HashSet不允许重复元素，可以为null)，保存到HashMap中的值是Object，HashSet提供的一些列函数实际上最终都是操作HashMap

遍历方式：

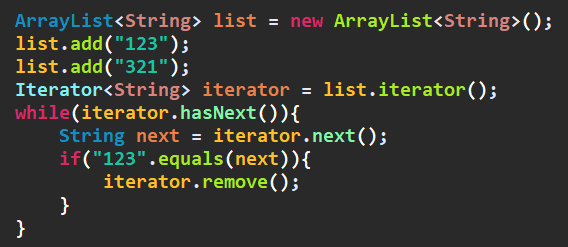
set.iterator()实际上遍历的HashMap的key

foreach

使用foreach的坑：



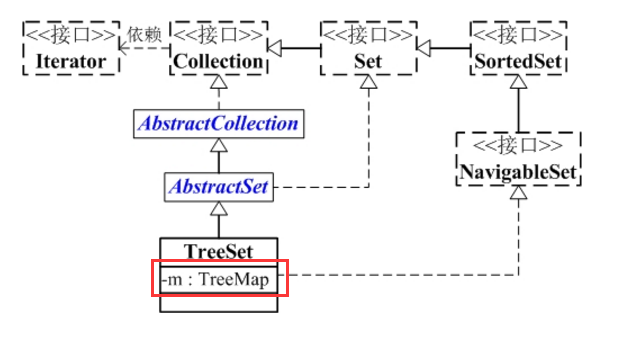
当执行remove的时候会抛出ConcurrentModificationException异常，因为foreach底层采用的是Iterator遍历，而Iterator的fail-fast机制，确保在遍历集合的时候，没有其他操作，执行remove后modCount会减一，而此时Iterator的modCount未变，所以会抛出异常(如果要remove，要调用Iterator的remove)

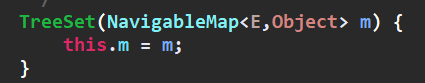
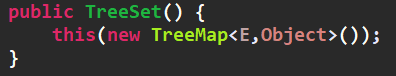


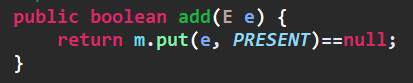
这样删除才不会抛异常，因为通过Iterator进行remove，Iterator会对modCount进行减一后，再去执行list.remove()

TreeSet：

TreeSet是一个有序的集合，和HashSet一样，TreeSet底层实际也是TreeMap



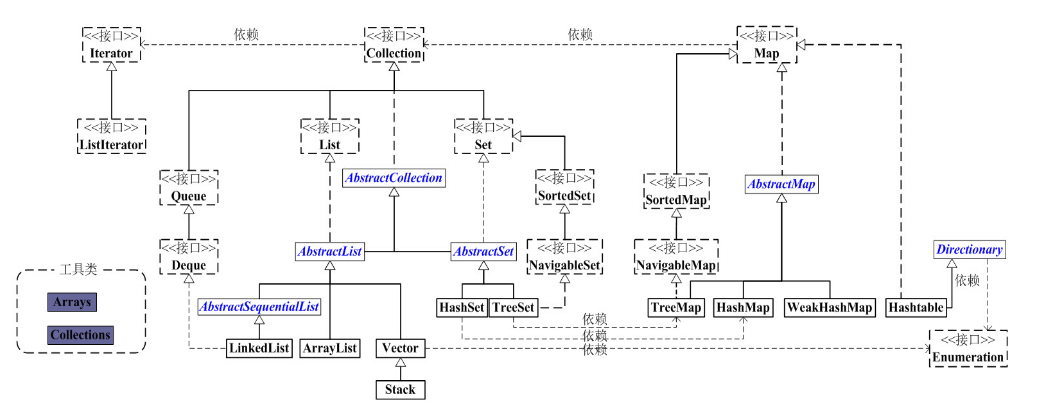




TreeSet的值是TreeMap的键，TreeMap的值是Object，所以TreeSet是有序的，TreeMap能操作的基本TreeSet也能

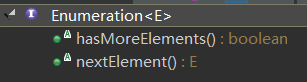
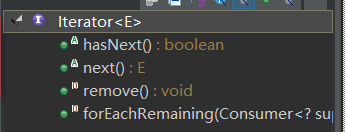
遍历：Iterator，foreach

LinkedHashSet基于LinkedHashMap



Iterator和Enumeration的区别：

Iterator和Enumeration都可以遍历集合，但是Iterator提供的方法更多



Iterator就是为了替代Enumeration的，效率上Iterator比Enumeration慢，因为Iterator实现了fail-fast机制

每个集合类中都有会实现一个自己的Iterator的子类，并且对Iterator的方法进行实现

Enumeration只能用于线程安全的集合，而非线程安全的采用Iterator的fail-fast机制确保安全

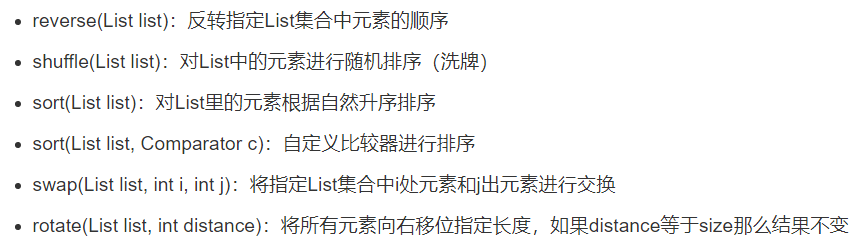
工具类：Collections/Arrays

Collections提供了很多操作Map和Collection的方法

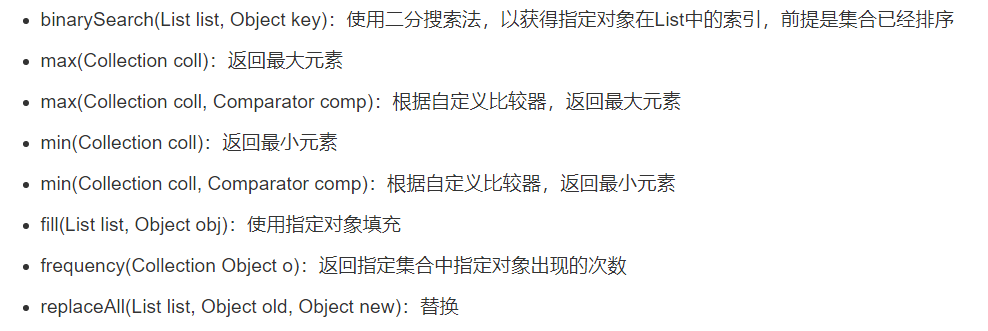
array类是java反射包下的一个类，用于判断类的属性是否是数组，而arrays是数组的一个工具类

常用的有：

针对List接口：

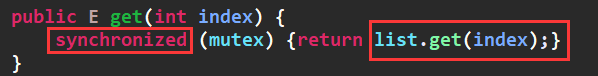


查找和替换(主要针对Collection接口)



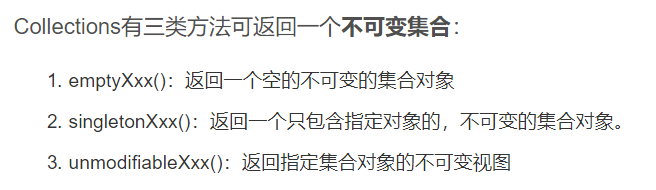
同步操作：Map和Collection中有些集合是非同步的可以通过Collections.synchronizedXxx将其同步(synchronizedSet、synchronizedSortedSet、synchronizedList、synchronizedMap、synchronizedSortedMap)





实现原理：返回一个synchronized的list执行方法时，在去调用实际list的方法，只不过加了一个锁，和Hashtable的实现方式类似

设置不可变集合：





返回的是一个空list不能对此list进行任何操作，调用add的话会报错(不可变)

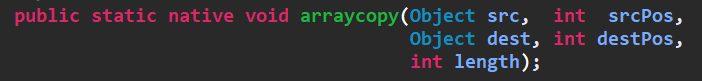
Arrays是对数组常见的工具类



Dog[] dog = new Dog[2]：在执行此语句时，会在堆空间创建两个内存地址块，用于保存Dog对象的引用，在数组保存的不是Dog而是Dog的引用

Arrays.copyof()和System.arraycopy()：

Arrays.copyof是操作集合的工具类，将集合转化为数组，但是实际底层操作的是System.arraycopy()，System.arraycopy()是将原来的数组复制到一个新的数组中



Native关键字表示此方法调用的是C/C++来完成的

src：源数组 srcPos：原数组复制的起始位置

dest：目标数组 destPos：目标数组放置的起始位置 length：复制的长度

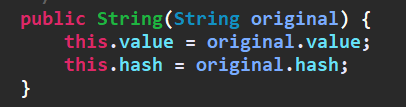
String：

String是一个final类，也就是String对象是不可变的，线程安全

不可变类即该类实例的属性是不可变的

线程安全：String类被创建后，就不会被修改了，即String本身就是线程安全的

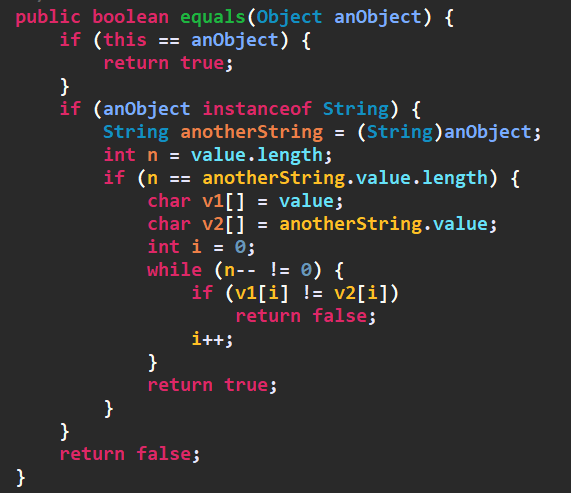
String内部包含了一个char数组来存放字符串(private final char value[])，private int hash：缓存String的hash值



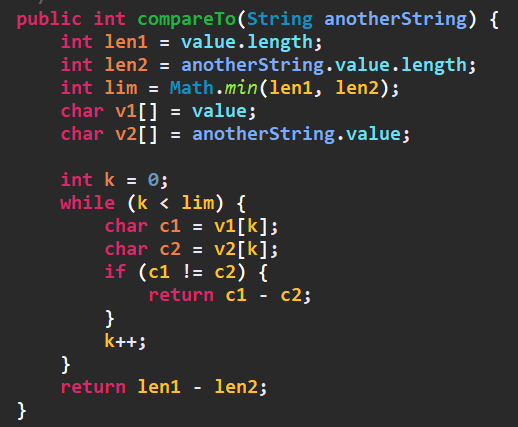
String提供了很多有参的构造器，但最终都是将结果保存到value数组中

String常用方法：

equals，先比较是否引用的是同一个对象，然后在一次比较值是否相等

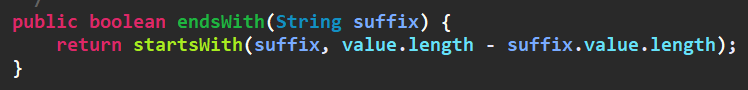


compareTo：返回值为0，两个字符串相等，否则不等



startsWith/endsWith：判断字符串开头/结尾是否是指定字符串

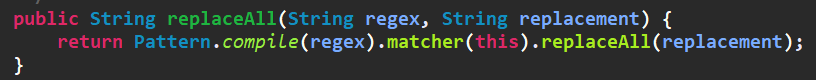




连接字符串：

将指定字符替换掉：

也可以通过正者表达式来替换字符串



trim()：去除字符串首尾的空格

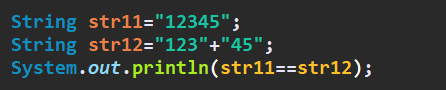
还有其他如getBytes，charAt(int)，indexOf(int)，subString(int)，split等常用方法

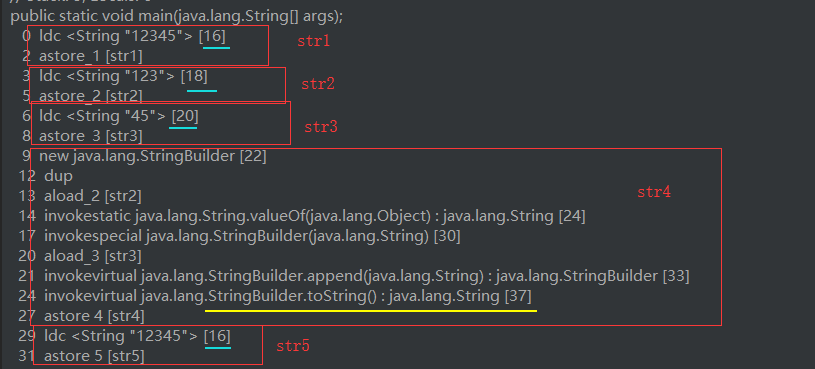
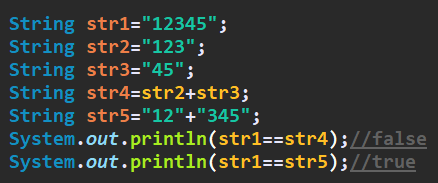
String str=String valueOf(12.63)：基本类型转换为字符串

String str=”123”和String str1=new String(“123”)的区别

当创建new String(“123”)会常见2两个对象，首先在常量池中创建一个123，然后在堆中创建一个String的实例对象(new String)，栈中的str1引用堆中的new String，newS tring指向常量池中的123，str1指向的地址是堆中new String的地址。而String str=”123”，先在常量池中创建123(前提是常量池在没有，如果有就不用出创建)，str指向的地址就是常量池中123的地址，String的==是比较的地址，equals比较的是字符是否相等

String str3=”12”+”3”的创建过程，先在常量池中声明12和3的地址，然后在创建一个常量123

输出结果为true



编译期和运行期：编译期是指将java文件转化为.class文件的过程，.class文件中都是一些指令，这些指令控制程序在运行的时候把内存分配好，分配的大小在编译的时候就知道了，而运行前分配内存则是说确定存放的位置。编译的时候，确定在内存中占有的大小，而运行期直接在内存中开辟出内存(堆，栈)即可

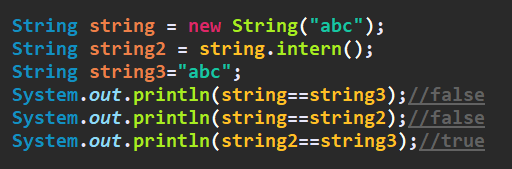
在编译期间会做一些简单的运算，比如这里的str5=”12”+”345”在编译的时候实际上就是str5=”12345”，str4=str2+str3是引用类型，是在运行的时候执行的，但是，在运行的时候，str4也应用是12345，但为啥不等？通过在Eclipse查看编译后的class文件，可以明白，在执行String+String的时候，会被编译为StringBuilder，然后调用append方法进行最加，最后是通过StringBuilder的toString方法返回str5，而toString方法实际返回的是一个new String(value, 0, count)，因此，str5是指向的是String对象

因为str2+str3是引用类型(和一般的int+int不一样，关键任然是StringBuilder返回的是String对象)，可能会变量，所以需要在运行时确定，而”12”+”345”是不会变的，所以在运行期确定(这里和编译期和运行期没多大关系，实际是String是引用类型，不能在编译器确定，而最终导致str1和str4不等，实际上是StringBuilder的原因)

编译期和运行期还体现在多态上：

Person p = new Man()：Person是Man的父类，通过父类指向子类的引用，在通过p输出属性时，实际是输出Person父类的值，而在输出方法的是Man子类的方法(继承后属性和方法)，在编译期，确定了p是Person，而在实际运行的时候p又指向的是子类Man，所以方法是执行子类的(这里记住就好，没必要深究)，如果想访问子类的属性((Man)p).属性名，但是不能访问父类的方法，因为无论p怎么变，在内存中实际只有Man一个对象，所以只能动态的访问Man的方法(类在加载的时候，会加载其属性及具体值，而方法是在运行的时候)，p还是一个类对象，类对象中只有属性

通过intern()获取字符串常量池中的值



intern返回的字符串常量的地址

intern的作用：判断常量池中是否有对应的字符串，如果有，返回此字符串的字面量的地址，，如果没有，在常量池中创建并返回字符串在常量池中的字面量地址

String虽然是线程安全的，但是每次都会在常量池中添加一些多余的字符串，消耗内存，而StringBuilder能够对字符进行动态的增删，但是StringBuilder也不是线程安全的，而StringBuffer是线程安全的，每个方法都使用了synchronized，StringBuffer是针对StringBuilder而言

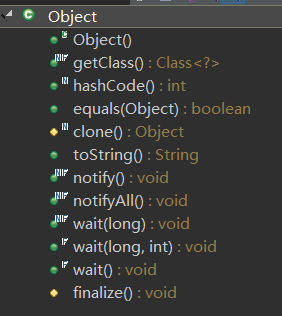
对于安全敏感的信息，使用char[]存储优于String(就是安全的信息的表示方式)：因为String是不可变的，长驻内存，直到GC，如果安全信息用String，那么信息将在内存在可见，不安全，所以使用可变的数组

JDK1.6以后，switch中可以使用字符串，之前只能使用int

String不可变的原因：final

String不可变的好处：String内部有hasecode值，java常使用hashcode进行比较，String不可变保证hashcode的一致性，在使用字符串的时候就不用在重新计算hashcode的值，不可变，安全，效率高

Object：



notify,notifyAll,wait是synchronized锁的操作

clone用户克隆当前对象，前提是当前对象必须实现Cloneable接口，否则调用clone方法时会抛异常

finalize()：当调用gc时，会执行此方法，垃圾回收机制执行之前做的一些操作，finally，final

length,length(),size()：

length是属性，用于获取数组的长度

length()是方法，用于获取字符串的长度

size()是方法，用于获取集合(Collection/Map)的长度

StringUtils是Apache的一个工具类在lang3中，常用方法有：

StringUtils.isEmpty(CharSequence cs)：判断字符序列是否为空/isNotEmpty

CharSequence是String/StringBuffer/StringBuilder的父接口

StringUtils.isAnyEmpty(“”,”123”,”321”)：只有有一个为空，就返回true/isNoneEmpty

StringUtils.isBlank(CharSequence cs)：判断字符串是否为空串(即是否有元素)/isNotBlank/isAnyBlank/isNoneBlank

isEmpty(“ ”);false/isBlank(“ ”);true

StringUtils.equals(cs1,cs2)：当为空时，不会报空指针异常/ equalsIgnoreCase

太多了，所有操作String的，都可以在StringUtils中找到对应的处理方式